B4M36DS2 – Database Systems 2

Lecture 6 – Key-Value stores: Redis

30. 10. 2023

Yuliia Prokop

prokoyul@fel.cvut.cz, Telegram **@Yulia_Prokop**

https://cw.fel.cvut.cz/b231/courses/b3b36prg/start

Outline

Examples

Publish / Subscribe

Geospatial

Transactions

RediSearch

RedisJSON

Persistence

Redis Architectures

EXAMPLES

Redis example - Banners

We want to place banners on the page at a certain position. We want to make them rotate evenly, that is, after each reload of the page these banners change.

ZADD banners **0** {banner}

Add a banner to the rotation

ZRANGE banners **1**

Will return a banner with fewer views

ZINCRBY banners 1 {banner}

Increase banner’s views

Redis example - Payment

Purchasing and payment

ZINCRBY balance 500 {User.id}

Top up balance

ZDECRBY balance 500 {User.id}

Withdraw the sum

ZADD purchases {Info}

Add information to a log

Redis example – Rating of the most popular

We want to show user activity ranking on the site – from the most active in descending order

ZREVRANGE user_rating 1 -1

Return the list in reverse order

ZRANGEBYSCORE user_rating 50 -1

Get everyone with more than 50 points

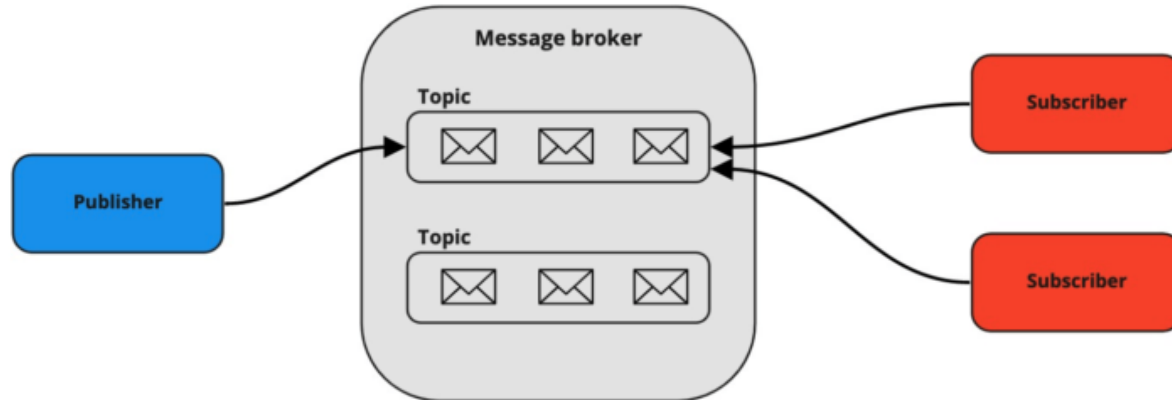
ZRANGEBYSCORE products 5000 10000

Get all the items in our online store that cost between 5k and 10k

ZRANGEBYSCORE logs timestamp1 timestamp2

Get logs accumulated for the period from timestamp1 to timestamp2

Publish / Subscribe



https://hevodata.com

Publish / Subscribe

For notifications and allerts

Redis is also a message broker that supports typical pub/sub operations.

- First user subscribes to certain channel "**news**"

```
SUBSCRIBE news
1) "subscribe"
2) "news"
3) (integer) 1
```

- Another user sends messages to the same channel "**news**"

```
PUBLISH news "hello"
(integer) 1
```

- Learn more at http://redis.io/commands#pubsub

GEOSPATIAL

Redis geospatial

Redis **geospatial** indexes let you store **coordinates** and search for them. This data structure is useful for finding nearby points within a given radius or bounding box.

GEOADD

adds a location to a given geospatial index
(note that longitude comes before latitude with this command).

GEOSEARCH

returns locations with a given radius or a bounding box.

GEODIST

returns the distance between two members in the geospatial index represented by the sorted set.

Redis example – Data format

For Redis geospatial commands, the correct format is longitude followed by latitude. Examples:

12.4964 41.9028

12.4964, 41.9028

The first number is the **longitude**, and the second is the **latitude**.

An option like

longitude 2.2945 latitude 48.8584

is not in the correct format for Redis geospatial commands because it includes additional text.

Redis example – Filter by location

GEOADD Addresses 43.361389 18.115556 "Addr1"
25.087269 37.502669 "Addr2"

GEODIST Addresses Addr1 Addr2
Distance between two addresses

GEOSEARCH Addresses FROMLONLAT 15 37 BYRADIUS 15 km ASC
Everything within a 15-kilometer radius of the point

More about geospatial: https://redis.io/docs/data-types/geospatial/

Transactions

Transaction

- All commands are serialized and executed sequentially
- Either all commands or no commands are processed
- Keys must be explicitly specified in Redis transactions
- Redis commands for transactions:
  - ✓ WATCH
    - Marks the given keys to be watched for conditional execution of a transaction.
  - ✓ MULTI
    - Marks the start of a transaction block. Subsequent commands will be queued for atomic execution using EXEC.
  - ✓ DISCARD
    - Flushes all previously queued commands in a transaction
  - ✓ EXEC
    - Executes all previously queued commands in a transaction
  - ✓ UNWATCH

Transaction - Example

MULTI
OK
INCR counter1
QUEUED
INCR counter1
QUEUED
DECR counter2
QUEUED
EXEC
1) (integer) 1
2) (integer) 2
3) (integer) -1

MULTI
OK
INCR counter1
QUEUED
INCR counter1
QUEUED
DECR counter2
QUEUED
DISCARD
OK

Transaction - Errors inside a transaction

- ✓ Before EXEC is called
  - The command may be syntactically wrong (wrong number of arguments, wrong command name, ...),
  - There may be some critical conditions like an out-of-memory condition.
- ✓ After EXEC is called
  - If we operated against a key with the wrong value (like calling a list operation against a string value).

- ❑ Redis does not support rollbacks of transactions.
- ❑ DISCARD can be used to abort a transaction. In this case, no commands are executed, and the state of the connection is restored to normal.

RediSearch

RediSearch

- ✓ RediSearch is a **Secondary Index** over Redis
  - Take a document
  - Break it apart
  - Map terms/properties and get a list of terms
  - Searching is getting documents that are linked to the terms
- ✓ **Full-Text** engine (Prefix, Fuzzy, Phonetic, Stemming, Synonyms…)
- ✓ **Incremental indexing** without performance loss
- ✓ Data **aggregation**
- ✓ Auto-complete **suggestions**
- ✓ **Geo** indexing and filtering

Iverted index

Document 1

(BSD licensed), in-memory data structure store used as a database, cache, message broker, and streaming engine. Redis

Document 2

You can run atomic operations on these types, like appending to a string; incrementing the value in a hash; pushing an element to a

Document 3

To achieve top performance, Redis works with an in-memory dataset. Depending on your use case, Redis can persist

Stopword list

a
achieve
also
an
and
appending
as
atomic
automatic
availability
bitmaps
broker
BSD
built-in
by
cache
can
case
Cluster
command
computing
data

Inverted index

1	a	1, 2, 3
2	achieve	3
3	also	3
4	an	1, 2, 3
5	and	1, 2
6	appending	2, 3
7	as	1
8	atomic	2
9	automatic	1
10	availability	1
11	bitmaps	1
12	broker	1
13	BSD	1
14	built-in	1
15	by	3
16	cache	1, 3
17	can	2, 3
18	case	3
19	Cluster	1
20	command	3
21	computing	2
22	data	1, 3

RediSearch - example

hset user:1 name "Anna" year "2000" friends "Tom, Michael, Helen"
hset user:2 name "Alex" year "1998" friends "Jakub, Helen"
hset user:3 name "Sandra" year "1999" friends "Jonas"

FT.CREATE usr_ind **prefix** 1 user: **SCHEMA** name **TEXT** year **NUMERIC** friends **TEXT**

Search for users, whose friend is Helen

FT.SEARCH usr_ind "Helen"

```
1) (integer) 2
2) "user:1"
3) 1) "name"
   2) "Anna"
   3) "year"
   4) "2000"
   5) "friends"
   6) "Tom, Michael, Helen"
4) "user:2"
5) 1) "name"
   2) "Alex"
   3) "year"
   4) "1998"
   5) "friends"
   6) "Jakub, Helen"
```

FT.SEARCH usr_ind "@friends:Helen"

```
1) (integer) 2
2) "user:1"
3) 1) "name"
   2) "Anna"
   3) "year"
   4) "2000"
   5) "friends"
   6) "Tom, Michael, Helen"
4) "user:2"
5) 1) "name"
   2) "Alex"
   3) "year"
   4) "1998"
   5) "friends"
   6) "Jakub, Helen"
```

Aggregation

- Perform a search query, filtering for records you wish to process.
- Build a pipeline of operations that transform the results by zero or more steps of:
  - **Group and Reduce**: grouping by fields in the results, and applying reducer functions on each group.
  - **Sort**: sort the results based on one or more fields.
  - **Apply Transformations**: Apply mathematical and string functions on fields in the pipeline, optionally creating new fields or replacing existing ones
  - **Limit**: Limit the result, regardless of sorting the result.
  - **Filter**: Filter the results (post-query) based on predicates relating to its values.

Aggregation - example

Log of visits to our website, each record containing the following fields/properties:

- **url** (text, sortable)
- **timestamp** (numeric, sortable) - unix timestamp of visit entry.
- **country** (tag, sortable)
- **user_id** (text, sortable, not indexed)

Select all records in the index, group the results by hour,
and count the distinct user IDs in each hour.
Then, format the hour as a human-readable timestamp

```
FT.AGGREGATE myIndex "*"
  APPLY "@timestamp - (@timestamp % 3600)" AS hour
  GROUPBY 1 @hour
    REDUCE COUNT_DISTINCT 1 @user_id AS num_users
  SORTBY 2 @hour ASC
  APPLY timefmt(@hour) AS hour
```

https://redis.io/docs/interact/search-and-query/search/aggregations/

RedisJASON

RedisJSON

The JSON capability of Redis Stack provides JavaScript Object Notation (JSON) support for Redis.

- Store, update, and retrieve JSON values.
- Index and query JSON documents.

✓ Full support for the JSON standard

✓ A **JSONPath** syntax for selecting/updating elements inside documents

✓ Documents are stored as binary data in a tree structure, allowing fast access to sub-elements

✓ Typed atomic operations for all JSON value types

JSON – Basic commands

JSON.SET key path value [NX | XX]

- Sets the JSON value at path in key
  - *Key* is a key to modify
  - *Path* is JSONPath to specify. The default is root $
  - *Value* is value to set at the specified path
  - *NX* sets the key only if it does not already exist
  - *XX* sets the key only if it already exists.

JSON.GET key [INDENT indent] [NEWLINE newline] [SPACE space] [path [path ...]]

- Returns the value at path in JSON serialized form

Example:
JSON.SET doc $ '{"a":2, "b": 3, "nested": {"a": 4, "b": null}}'
OK
JSON.GET doc $..b
"[3]"

RedisJSON - Example

JSON.SET item:1 $ '{"**name**":"Noise-cancelling Bluetooth headphones",

"**description**":"Wireless Bluetooth headphones with noise-cancelling technology",

"**connection**":{"wireless":true,"type":"Bluetooth"},"**price**":99.98,"**stock**":25,

"**colors**":["black","silver"],"**embedding**":[0.87,-0.15,0.55,0.03]}'

OK

JSON.SET item:2 $ '{"**name**":"Wireless earbuds",

"**description**":"Wireless Bluetooth in-ear headphones",

"**connection**":{"wireless":true,"type":"Bluetooth"},"**price**":64.99,"**stock**":17,

"**colors**":["black","white"],"**embedding**":[-0.7,-0.51,0.88,0.14]}'

OK

https://redis.io/docs/interact/search-and-query/indexing/

RedisJSON - Indexing

```
FT.CREATE {index_name}
ON JSON SCHEMA {json_path} AS {attribute} {type}
```

Example:

Create an index that indexes the name, description, price, and image vector embedding of each JSON document that represents an inventory item

FT.**CREATE** itemIdx
ON JSON **PREFIX** 1 item: **SCHEMA** $.name **AS** name TEXT $.description as description TEXT $.price AS price NUMERIC $.embedding AS embedding VECTOR FLAT 6 DIM 4 DISTANCE_METRIC L2 **TYPE** FLOAT32

https://redis.io/docs/interact/search-and-query/indexing/

RedisJSON - Search the index

Search for earbuds:

FT.**SEARCH** itemIdx '@name:(**earbuds**)'

1) (integer) 1

2) "item:2"

3) 1) "$"

   2) "{\"name\":\"Wireless **earbuds**\",
\"description\":\"Wireless Bluetooth in-ear headphones\",
\"connection\":{\"wireless\":true,\"type\":\"Bluetooth\"},
\"price\":64.99,\"stock\":17,\"colors\":[\"black\",\"white\"],
\"embedding\":[-0.7,-0.51,0.88,0.14]}"

https://redis.io/docs/interact/search-and-query/indexing/

RedisJSON - Search the index

Search for Bluetooth headphones with a price of less than 70:

FT.SEARCH itemIdx '@**description**:(bluetooth headphones) @**price**:[0 70]'

1) (integer) 1

2) "item:2"

3) 1) "$"

   2) "{\"name\":\"Wireless earbuds\",

   \"description\":\"Wireless Bluetooth in-ear headphones\",

   \"connection\":{\"wireless\":true,\"type\":\"Bluetooth\"},\"price\":64.99,\"stock\":17,

   \"colors\":[\"black\",\"white\"],\"embedding\":[-0.7,-0.51,0.88,0.14]}"

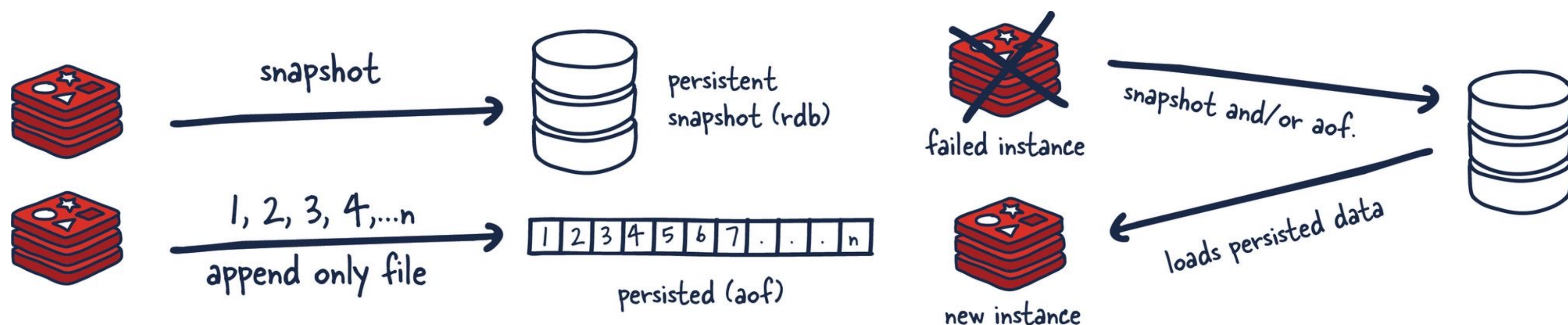
Read more:
https://redis.io/docs/interact/search-and-query/query/
https://redis.io/docs/interact/search-and-query/indexing/

PERSISTENCE

Persistance

Datasets can be saved to disk

Persistence refers to the writing of data to durable storage, such as a solid-state disk (SSD).



Source: https://architecturenotes.co/redis/

Persistence

Redis provides a range of persistence options. These include:

- **RDB** (Redis Database): RDB persistence performs point-in-time snapshots of your dataset at specified intervals.
- **AOF** (Append Only File): AOF persistence logs every write operation the server receives. These operations can then be replayed at server startup, reconstructing the original dataset.
- **No persistence**: You can disable persistence completely. This is sometimes used when caching.
- **RDB + AOF**: You can combine AOF and RDB in the same instance.

RDB Example

Example 1: RDB Persistence with Custom Save Points

In **redis.conf**, set the following options

save 900 1

Save the DB if at least 1 key changed in 900 seconds

save 300 10

Save the DB if at least 10 keys changed in 300 seconds

save 60 10000

Save the DB if at least 10000 keys changed in 60 seconds

AOF Example

Example 2: AOF Persistence with Every Second fsync

In **redis.conf**, set the following options

appendonly yes

    # Enable AOF persistence

appendfsync everysec

    # fsync every second

RDB and AOF Example

Example: RDB + AOF Persistence with No fsync

In **redis.conf**, set the following options

save 3600 1

  # Save the DB if at least 1 key changed in 3600 seconds

appendonly yes

  # Enable AOF persistence

appendfsync no

  # Do not fsync, leave it to the OS

RDB advantages and disadvantages

- ✓ RDB is a very compact single file for backups and disaster recovery.
- ✓ RDB maximizes Redis performances since the only work the Redis parent process needs to do to persist is forking a child that will do all the rest. The parent process will never perform disk I/O or alike.
- ✓ RDB allows faster restarts with big datasets compared to AOF.

- **Data Loss**: RDB snapshots are taken periodically, which means that you could lose data not yet included in the most recent snapshot in case of a system crash.
- **Forking Overhead**: The Redis process needs to fork a child process to create the RDB snapshot, which can be resource-intensive for large datasets.

AOF advantages and disadvantages

- ✓ **Better Durability**: AOF provides better data durability, as it logs every write operation, reducing the risk of data loss.
- ✓ **Human-Readable Format**: AOF files store the commands in a plain text format, making them easy to inspect and understand.
- ✓ **Flexible Configuration**: You can configure the AOF fsync policy to balance durability and performance based on your requirements.

Disadvantages of AOF

- **Larger File Size**: AOF files can be significantly larger than RDB files, as they store every write operation.
- **Slower Recovery**: The recovery process for AOF can be slower than RDB, as Redis needs to replay all the logged commands to reconstruct the dataset.

Persistence: RDB vs AOF

RDB (Redis Database File)	AOF (Append Only File)
Provides point in time snapshots	Logs every write
Creates complete snapshot at specified interval	Replays at server startup. If log gets big, optimization takes place
File is in binary format	File is easily readable
On crash minutes of data can be lost	Minimal chance of data loss
Small files, fast (mostly)	Big files, 'slow'

Source: https://www.slideshare.net/MaartenSmeets1/introduction-redis-93365594

RDB related commands

BGSAVE

Save the DB in the background. Redis forks, the parent continues serving the clients, and the child saves the dataset on disk and exits.

SAVE

Perform a synchronous save of the dataset. Other clients are blocked – never use in production!

LASTSAVE

Return the Unix time of the last successful DB save.

AOF related commands

BGREWRITEAOF

Instruct Redis to start an AOF rewrite process. The rewrite will create a small optimized version of the current AOF log.

If **BGREWRITEAOF** fails, no data gets lost, as the old AOF will be untouched

What persistence is used for?

- ✓ **Backups**
- ✓ **Disaster Recovery**
- ✓ **Performance Maximization**
- ✓ **Faster Restarts with Big Datasets**
- ✓ **Replicas**

Redis Architectures

Redis Architecture

Simple Database

Main

HA Database

Main replication Secondary

Redis sentinel

Sentinel nodes

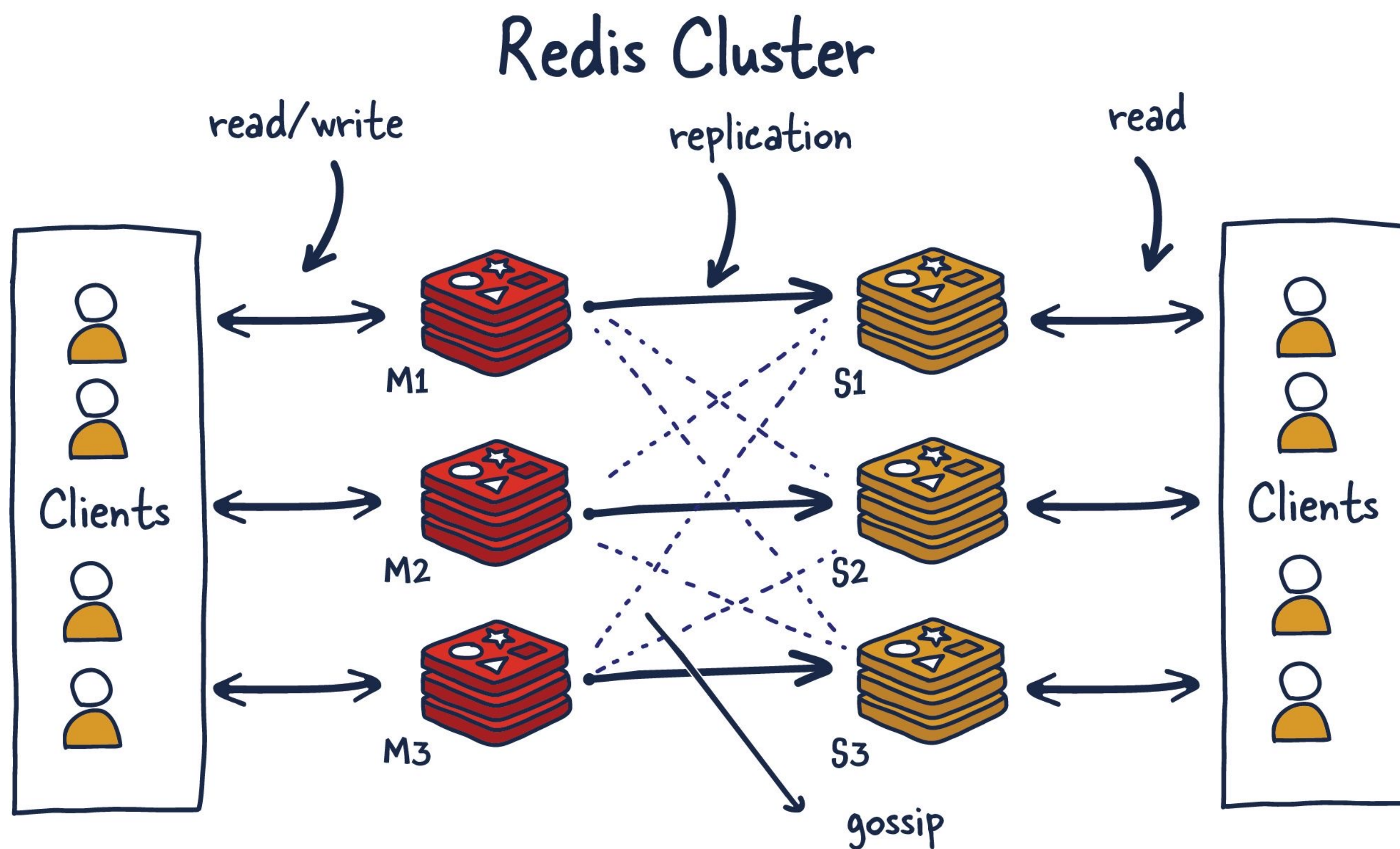
Main

replication

Secondary 1 Secondary 2

Source: https://architecturenotes.co/redis/

Redis Architecture



Source: https://architecturenotes.co/redis/

Redis and the CAP Theorem

Availability:

- Redis uses a master-slave replication model to ensure high availability.
  - There is a single "master" node that accepts all writes and multiple "slave" nodes that replicate data from the master in real-time.
  - In the event of a failure of the master node, one of the slave nodes can be promoted to become the new master.

Consistency:

- Redis provides strong consistency guarantees for single-key operations.
  - If a value is written to a key, it will be immediately available for reads from any node in the cluster.
  - However, Redis does not provide transactional consistency for multi-key operations, meaning that it is possible for some nodes to see a different view of the data than others.

Partitioning:

- Redis supports sharding, which allows the data set to be partitioned across multiple nodes.
  - Redis uses a hash-based partitioning scheme, where each key is assigned to a specific node based on its hash value.
  - Redis also provides a mechanism for redistributing data when nodes are added or removed from the cluster.

Summary. Why Redis?

- ✓ **In-Memory Data Storage**
- ✓ **Data Structure Support**
- ✓ **Persistence Options**
- ✓ **Pub/Sub Messaging**
- ✓ **Caching**
- ✓ **Distributed Architecture**
- ✓ **Extensibility**

Summary

1. Redis is ultra-fast in-memory data store
   - Not a database, used along with databases
2. Supports strings, numbers, lists, hashes, sets, sorted sets, publish / subscribe messaging
3. Used for caching / simple apps

Redis is a powerful tool for system design, but it may not be suitable for all use cases. It is important to carefully consider its limitations when deciding whether to use Redis in a particular application.